Reversing with Radare2

Starting Radare

The basic usage is `radare2` *exe* (on some systems you can use simply `r2` instead of `radare2`). If there exists a script named *exe*`.r2`, then it gets executed after the others rc-files. If you want to run radare2 without opening any file, you can use `--` instead of an executable name.

Some command-line options are:

`-d` *file*	debug executable *file*
`-d` *pid*	debug process *pid*
`-A`	analyze all referenced code (`aaa` command)
`-r` *profile*`.rr2`	specifies `rarun2` profile (same as `-e dbg.profile=`*profile*`.rr2`)
`-w`	open file in write mode
`-p` [*prj*]	list projects / use project *prj*
`-h`	show help message (`-hh` the verbose one)

Example: `r2 -dA /bin/ls`

Running in different environments: rarun2

`rarun2` runs programs with different environments, arguments, permissions, directories and overridden default file-descriptors. Usage:
`rarun2` [`-t`|*script-name*`.rr2`] [*directives*] [`--`] [*prog-name*] [*args*]
`rarun2 -t` shows the terminal name, say α, and wait for a connection from another process. For instance, from another terminal, you can execute `rarun2 stdio=`α `program=/bin/sh` (use `stdin`/`stdout` to redirect one stream only). Run `rarun2 -h` to get a sample `.rr2` file. rarun2 supports *a lot* of directives, see the man page for details.

General information

The command `?` prints the help. Command names are hierarchically defined; for instance, all **p**rinting commands start with `p`. So, to understand what a command does, you can append `?` to a *prefix* of such a command; e.g., to learn what `pdf` does, you can first try `pd?`, then the more general `p?`. You can get recursive help with `?*`; e.g.: `p?*`
Single-line comments can be entered using `#`; e.g. `s # where R we?`.
Command `?` can also be used to evaluate an expression and print its result in various format; e.g. `? 5 * 8+2` (note the space after `?`).
Commands `?v`/`?vi` print result only in hex/decimal. There are also some special `$`-variables (list them all with: `?$?`); e.g.:

`$$`	current virtual seek
`$b`	block size

Where an address *addx* is expected, you can provide any expression that evaluates to an address, e.g. a function name or a register name. In this cheatsheet we sometimes use *fn-name*, instead of *addx*, to emphasize that the argument is supposed to be a function starting address. As default address is (usually?) used the current seek: `$$`.
All commands that:

- accept an optional size (e.g. `pd`), use the current block size by default (see: `b`)
- accept an optional address (e.g., `pdf`), use the current position by default (see: `s`)

Commands can be chained by using `;`; e.g. `s fun; pd 2`.
A single command can be applied to each element of a sequence by using `@@`; e.g. `axt @@ str.*`, see `@@?`.

Internal grep-like filtering

You can filter command output by appending `~`[`!`]*str*, to display only rows [not] containing string *str*; e.g. `pdf~rdx` and `pdf~!rdx`. You can further filter by appending

`:`*r*	display row *r* ($0 \le r < \#rows$ or, backwards with: $-\#rows \le r \le -1$)
`[`c_1`[,`c_2`,...]]`	display columns $c_1, c_2, \ldots$ ($0 \le c_i < \#cols$)
`:r[`$c_1, \ldots, c_n$`]`	display columns $c_1, \ldots, c_n$ of row *r*
`..`	pipe output into less-like viewer
`...`	pipe into HUD, which filters space separated strings

Examples: `afl~[0]`, `afl~malloc[0]`, `pdf~:2` and `pdf~mov:2`
There is much more (sorting, counting, ...); see: `~?`

Shell interaction

Command output can be redirected to a file by appending `>`*filename* or piped to an external command with `|`*progname* [*args*]. Examples: `afl > all_functions` and `afl | wc -l`.
External commands can be run with `!!`*progname* [*args*]. Note: if a command starts with a single `!`, the rest of the string is passed to currently loaded IO plugin (only if no plugin can handle the command, it is passed to the shell).
Moreover, backticks can be used to send the output of r2-commands as arguments; e.g. `!!echo ‘? 42‘`. Vice versa output of external programs can be used as arguments for internal commands; e.g. `pdf ‘echo 3‘ @ ‘echo entry0‘`.
Some common Unix-like commands are implemented as built-ins; e.g. `ls`, `cd`, `pwd`, `mkdir` and `rm`.

Radare scripting

`.` *filename*	interpret r2 script *filename*
`.!` *command*	interpret output of *command* as r2 commands

Python scripting (via r2pipe)

You can script Radare2 with Python, by leveraging *r2pipe*, that can be easily installed (inside any Python 2 virtual environment) with: `pip install r2pipe`.
Then, you can spawn a Python interpreter, from inside r2, with: `#!pipe python` [*python-file*]
or simply:
`#.` *python-file*
Once you are in Python-world, you can connect to r2 by `importing` `r2pipe` and inizializing some variable, say `r2`, with `r2pipe.open("#!pipe")`, or simply `r2pipe.open()`.
Then you can interact with Radare by invoking method `cmd`; e.g. `print(r2.cmd('pdf @ entry0'))`.
You can make most Radare2 commands output in JSON format by appending a `j`; e.g. `pdfj` (instead of `pdf`).
Method `cmdj` can de-serialize JSON output into Python objects; e.g.

```
f = r2.cmdj('pdfj @ entry0')
print f['name'], f['addr'], f['ops'][0]['opcode']
```

r2pipe: connecting to other r2 instances

You can connect to any web-listening instance of r2 by passing `r2pipe.open` a string of the form `'http://`*host*`:`*port*`'`. By using this approach you get your own seek-cursor: your seek commands won't affect others.

To open a background web-service in r2 use command `=h&`. You may also want to take a look at configuration variable `http.sandbox`.

Configuration

`e??`	list all variable names and descriptions
`e?`[`?`] *var-name*	show description of *var-name*
`e` [*var-name*]	show the value of all variables [*var-name* only]
`e` *var-name* `=?`[`?`]	print valid values of *var-name* [with descript.] E.g. `e asm.arch=??`
`eco` *theme-name*	select theme; eg. `eco solarized`
`eco`	list available themes
`b` [*size*]	display [set] current block size
`env` [*name* [`=`*value*]]	get/set environment variables

Some variables

`asm.pseudo`	enable pseudo-code syntax
`asm.bytes`	display bytes of each instruction
`asm.describe`	show opcode description
`asm.cmtright`	comments at right of disassembly if they fit
`asm.emu`	run ESIL emulation analysis on disasm
`asm.demangle`	Show demangled symbols in disasm
`bin.demangle`	Import demangled symbols from RBin
`cmd.bp`	command to run when a breakpoint is hit; e.g. `cmd.bp=!!`*program*
`cmd.stack`	command to display the stack in visual debug mode (Eg: `px 32`)
`dbg.follow.child`	continue tracing the child process on fork
`dbg.slow`	show stack and regs in visual mode, in a slow but verbose (e.g. telescoping) mode; check *column mode*
`dbg.trace`	trace program execution (check also `asm.trace`)
`io.cache`	enable cache for IO (=non-persistent write-mode)
`scr.utf8`	show nice UTF-8 chars instead of ANSI (Windows: switch code-page with `chcp 65001`)
`scr.utf8.curvy`	show curved UTF-8 corners (requires `scr.utf8`)
`scr.nkey`	select seek mode; affects `n`/`N` in visual mode
`scr.breaklines`	break lines in Visual instead of truncating them
`scr.html`	disassembly outputs in HTML syntax
`scr.wheel`	enables mouse-wheel in visual mode

Searching: `/`

`/` *str*	search for string *str*
`/x` *hstr*	search for hex-string *hstr*
`/a` *asm-instr*	assemble instruction and search for its bytes
`/R`[`/`] *opcode*	find ROP gadgets [with r.e.] containing opcode; see: `http://radare.today/posts/ropnroll/`
`/A` *type*	find instructions of type *type* (`/A?` for the listof types)

Also: `e search.in=??` and `e??search` for options

Seeking: `s`

`s`	print current position/address
`s` *addx*	seek to *addx*
`s..` *hex*	changes least-significant part of current address to *hex*
`s+` *n* and `s-` *n*	seek *n* bytes forward/backward
`s++` and `s--`	seek block-size bytes forward/backward
`s-`	undo seek
`s+`	redo seek
`s=`	list seek history
`s*`	list seek history as r2-commands

Writing: w

wa *asm-instr*	assemble+write opcodes; quote the whole command for more instructions: "wa *instr*$_1$; *instr*$_2$; ..."
wao ...	replace current instruction; see wao? for details
w[z] *str*	write string *str* [and append byte \x00]
wx *hex-pairs*	write hex-pairs
wc	list pending changes (see variable io.cache)
wtf [*file*] [*size*]	write to file
wopO *v*	print offset of *v* inside De Bruijn pattern; equiv. to ragg2 -q *v*; to produce a pattern: ragg2 -r -P *size*

Analysis (functions and syscalls): a

aaa	analyze (aa) and auto-name functions
afl[l]	list functions [with details]
afi *fn-name*	show verbose info for *fn-name*
afn *new-name addx*	(re)name function at address *addx*
asl	list syscalls
asl *name*	display syscall-number for *name*
asl *n*	display name of syscall number *n*
afvd *var-name*	output r2 command for displaying the address and value of arg/local *var-name*
.afvd *var-name*	display address and value of *var-name*
afvn *name new-name*	rename argument/local variable
afvt *name type*	change type for given argument/local
afv- *name*	removes variable *name*
axt *addx*	find data/code references to *addx*
ahi {b|d|h|o|r|S|s} @ *addx*	define binary/decimal/hex/octal/IP/ syscall/string base for immediate

ESIL: ae

aeim	initialize ESIL VM stack
aepc *addr*	change ESIL PC to *addx* (aeip sets PC to curseek)
aer ...	handle ESIL registers like dr does
aes[b|o]	perform emulated debugger step [back|over]
aesu *addr*	step until given address

Graphviz/graph code: ag

ag *addr*	output graphviz code (BB at *addr* and children) E.g. view the function graph with: ag $$ | xdot -
agc *addr*	callgraph of function at *addx*
agC	full program callgraph

Flags (AKA "bookmarks"): f

fs [*name*]	display flagspaces [select/create fs *name*]
fs+ *name*	push previous flagspace and set *name*
fs-	pop to the previous flagspace
f	list flags
f *name* @ *addx*	*or*
f *name* = *addx*	associate name *name* to address *addx*
f- @ *addx*	remove the association at address *addx*
f- *name*	remove the association with name *name*

Comments: C

CCu *text* [@ *addx*]	set (update?) comment *text* at *addx*
CC *text* [@ *addx*]	append comment *text* at *addx*
CC- [@ *addx*]	remove comment at *addx*
CC. [@ *addx*]	show comment at *addx*
CC! [@ *addx*]	edit comment using cfg.editor (vim, ...)

Debugging: d

?d *opcode*	description of *opcode* (eg. ?d jle)
dc	continue (or start) execution
dcu *addx*	continue until *addx* is reached
dcs [*name*]	continue until the next syscall [*name*]
dcr	continue until ret (uses step over)
dr=	show general-purpose regs and their values
dro	show previous (old) values of registers
drr	show register references (telescoping)
dr *reg-name* = *value*	set register value
drt	list register types
drt *type*	list registers of type *type* and their values
db	list breakpoints
db[-] *addx*	add [remove] breakpoint
doo [*args*]	(re)start debugging
ood	synonym for doo
ds[o]	step into [over]
dbt	display backtrace (check dbg.btdepth/btalgo)
drx	hardware breakpoints
dm	list memory maps; the asterisk shows where the current offset is
dmm	list modules (libraries, loaded binaries)
dmi [*addr*|*lib*] [*sym*]	list symbols of target lib
dmp	change page permissions (see: dmp?)
dt[d]	list all traces [disassembled]

Types: t

"td *C-type-def*"	define a new type
t *t-name*	show type *t-name* in pf syntax
.t *t-name* @ *addx*	display the value (of type *t-name*) at *addx*
t	list (base?) types
te / ts / tu	list enums/structs/unions
to *file*	parse type information from C header file
tl *t-name*	link *t-name* to current address
tl *t-name* = *addx*	link *t-name* to address *addx*
tl	list all links in readable format
tp *t-name* = *addx*	cast data at *addx* to type *t-name*, and prints it

Printing: p

ps [@ *addx*]	print C-string at *addx* (or current position)
pxr [*n*] [@ *addx*]	print with references to flags/code (telescoping)
px [*n*] [@ *addx*]	hexdump — note: x is an alias for px
px{h|w|q} ...	hexdump in 16/32/64 bit words
px{H|W|Q} ...	as the previous one, but one per line
pxl [*n*] [@ *addx*]	display *n* rows of hexdump
px/*fmt* [@ *addx*]	gdb-style printing *fmt* (in gdb see: help x from r2: !!gdb -q -ex 'help x' -ex quit)
pd [*n*] [@ *addx*]	disassemble *n* instructions
p8 [*n*] [@ *addx*]	print bytes
pD [*n*] [@ *addx*]	disassemble *n* bytes
pd -*n* [@ *addx*]	disassemble *n* instructions backwards
pdf [@ *fn-name*]	disassemble function *fn-name*
pc[p] [*n*] [@ *addx*]	dumps in C [Python] format
* *addx* [=*value*]	shortcut for reading/writing at *addx*
pf *fmt* a_1[, a_2, ...]	formatted print, see pf?? and pf???

Information: i (and S)

i	show info of current file
iz[z]	strings in data sections [whole binary]
i{e|i|l|S}	entrypoint/imports/libraries/sections
S	list segments (confusingly called sections?!?)

Visual mode: V (q exits)

Command V enters *visual mode*.

q	exit visual-mode
c	cursor-mode, *tab* switches among panels
	+/- increment/decrement current byte
:	execute a normal-mode command; e.g. :dm
p and P	rotate forward/backward print modes
/*str*	highlight occurrences of string *str*
$	toggle pseudo-syntax
O	toggle ESIL-asm
;	add/remove comments (to current offset)
x	browse xrefs-to current offset
X	browse xrefs-from current function
-	browse flags
d	define function, end-function, rename, ...
di{b|o|d|h|s}	define immediate bin/oct/dec/hex or str
V	enter block-graph viewer (*space* toggles visual/graph)
A	enter visual-assembler (preview must be confirmed)
n / N	seek next/previous function/flag/hit (see scr.nkey)
i	enter insert mode
e	configures internal variables
"	toggle the column mode

Seeking (in Visual Mode)

.	seeks to program counter
Enter	on jump/call instructions, follow target address
u / U	undo / redo
o	go/seek to given offset
0 (*zero*)	seek to beginning of current function
d (a non-zero digit)	jump to the target marked [*d*]
m*l* (a letter)	mark the spot with letter *l*
'*l*	jump to mark *l*
n / N	jump to next/previous function

Debugging (in Visual Mode)

b or F2	toggle breakpoint
F4	run to cursor
s or F7	step-into
S or F8	step-over
F9	continue

Projects: P [unstable feature]

Pl	list all projects
P{o|s|d} [*prj-name*]	open/save/delete project *prj-name*
Pc *prj-name*	show project script to console